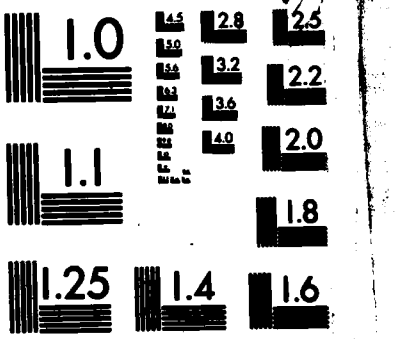
AD-A140 780 A HIGH PERFORMANCE BULK MEMORY SYSTEM(U) YALE UNIV NEW HAVEN CT DEPT OF COMPUTER SCIENCE W GROPP ET AL. 1/1
MAR 84 YALEU/DCS/RR-311 N00014-82-K-0184

UNCLASSIFIED F/G 9/2 NL



13

A High Performance Bulk Memory System

William Gropp, John J. O'Donnell, Susan Temple O'Donnell,
Martin H. Schultz, and Brian Weston

Research Report YALEU/DCS/RR-311
March 1984

YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

A High Performance Bulk Memory System

William Gropp, John J. O'Donnell, Susan Temple O'Donnell,
Martin H. Schultz, and Brian Weston

Research Report YALEU/DCS/RR-311
March 1984

This work was supported in part by ONR Grant #N00014-82-K-0184, NSF Grant #MCS-8106181, and external Research Grants from Digital Equipment Corporation and Floating Point Systems.

1. Introduction

The scale of computations in applications as diverse as commercial data processing, data base systems, engineering and scientific simulations, CAD/CAM systems, and artificial intelligence systems is increasing very rapidly. Moreover, with the growing power of modern supercomputers, this increase will probably accelerate in the next decade. As the data sets for these large computations increase, there is a common computational bottleneck in all these applications: the transfer of data between the primary and secondary memories of computer systems. The inner loops of many computer programs do few operations per datum, and the speed of the microelectronic components of central processing units and memories of modern computers greatly exceeds the transfer rates to and from (mechanically based) secondary memories. Consequently, a computer system will often spend all its time shuffling data back and forth between primary and secondary memories. As a result, many large computations on high performance computer systems realize a very small percentage of the available resource of the CPU.

With parallel computation dramatically increasing the available computing power, this problem will only become worse. We need a significant improvement in the technology of secondary memories. A bulk memory system with inexpensive memory doing fetches or writes in parallel (interleaving) provides one such technology. This interleaving is perfect for scientific computing in which one can usually design algorithms so that one need only access very long vectors in auxiliary storage. In these applications the effectiveness of the bulk memory is determined by the transfer rate. However, the bulk memory is even better suited for those problems, for example, as arise in commercial data processing, data base, and artificial intelligence applications, in which we wish to transfer small amounts of data. In these applications the effectiveness of the bulk memory is determined not by the transfer rate but by the latency and access time. Our data, in Section 5, will show that the relative improvement of the latency and access time over mechanical secondary storage devices is greater than the relative improvement in the transfer rates. We believe that the availability of such secondary storage devices is likely to promote the development of new algorithms that can make effective use of transfers to large-scale bulk memories. Quite possibly such algorithms will be much more efficient in execution speed than current algorithms which are designed to minimize the use of secondary storage.

In order to test the validity of our thesis, we have developed a bulk memory secondary storage device for a general-purpose supercomputer, the Floating Point Systems FPS-164. We present an overview of of this system in Section 2. In Section 3, we describe the design philosophy of

the interface to the FPS-164. To simplify the use of the bulk memory system, we have written a software system for the bulk memory which emulates a disk. This enables the user to use application software written to do I/O on a system with a peripheral disk virtually unchanged. We present the details of our software design in Section 4. The FPS-164 is a moderately priced, fast central processing unit with fast I/O ports. While array processors such as the FPS-164 have traditionally been used for engineering and scientific computations, we feel the lessons we learn with this system will carry over to other applications and other high performance computers. We view the existing system and experiments reported in Section 5, to be a "proof of concept." As we will show, our bulk memory provides a significant improvement to system performance for computations which are I/O intensive. Moreover, the improvement comes at a relatively modest cost.

2. Overview of the Bulk Memory System

The FPS-164 is an 11 megaflop, 64 bit processor with an excellent optimizing FORTRAN compiler. As such, it makes a fast, inexpensive backend processor for such compute intensive applications as CAD/CAM. The FPS-164 is 10 to 20 times less expensive than traditional mainframe supercomputers and it delivers about one-tenth the speed. Perhaps more important for our purpose, the FPS-164 has very high speed, 44 MB/second, I/O ports.

However, for I/O intensive computations, standard mechanical auxiliary storage devices (disks) are too slow and significantly degrade the performance of the system. We have worst case examples in which a computation requires about three seconds but the I/O for the computation requires three hundred seconds. Such examples are common in large scale I/O intensive computations. There are two remedies to this difficulty. The traditional remedy, promoted strongly by FPS, is to buy enough main memory so that the computation runs in main memory and the auxiliary storage is unnecessary. This remedy has a couple of flaws:

1. No matter how much main memory one puts on a system there are applications which require more.
2. Main memory on the FPS-164 is very expensive (about $20,000/MB).

Our remedy is to put a large, inexpensive, fast bulk store on the machine.

Yet, despite its great power, the bulk memory is no more difficult to use than a disk. In fact, we have written system software that does a disk emulation for the bulk memory even to the point of allowing permanent user files, cf. Section 4 for details.

The current implementation of the Yale bulk memory system allows up to six boxes, each containing a maximum of 32 MB of memory and two I/O ports, to be chained together (at the cost of one port per box) to yield a memory system with a maximum of megabytes and six ports. The current system is designed so that its memory chips can be upgraded to 256K memory chips when they are available, which will yield a system with a 768 MB capacity. The total bandwidth of the memory system is 64 MB/second for vector transfers and .6 MB/second for scalar transfers. See Section 3 for details about our interface design philosophy.

We are thoroughly benchmarking the bulk memory system. Our results which we present in Section 5, are very impressive. We have seen benchmarks in which the FPS-164 with the bulk memory system is four times faster than a Cray-1 using its disk for auxiliary storage.

3. Details of Hardware Implementation

3.1. Subsystem design goals

Our design goals for the memory subsystem were as follows:

- It should look to the FPS-164 as an I/O device. Attempts to transparently extend the main-memory address space would be less likely to use the unique characteristics of the memory we were working with, would tie us too closely to the implementation details of the FPS-164, and would be unlikely to scale to other computer systems or teach us lessons that would generalize.
- It should use the densest, cheapest monolithic memory technology. Nothing is served by building a memory that duplicates the cost of primary memory, or the speed of traditional secondary stores. We aimed to plug an important gap in memory hierarchies in practical systems.
- It should be able to run sequential block transfers at the highest possible rate. If necessary we were willing to lengthen transfer-start latency in pursuit of this goal.
- The subsystem should be expandable to large capacity. This implies cost and reliability goals, as well as requiring us to handle multiple memory backplanes.
- The subsystem should support experimentation in multilevel memory hierarchies, allowing us to attach slower, larger memories such as disks without moving the data back through the FPS-164. We are interested in studying "disk cache" and "staging memory" techniques for very large problems.
- The system should be as off-the-shelf and as reliable as possible.

3.2. What we started with

A number of vendors supply memory subsystems packaged with controllers. At the time we undertook the project, only Motorola and Dataram offered interfaces that allowed the user a wide word interface and control over interleaving to achieve the required throughput, and only Motorola had actual field installations of their product, now known as the Zitel System/3000.

The System/3000 transfers data at 64MB/second when in sequential-access mode, with approximately 1μs latency in word-at-a-time mode. The subsystem includes error correction and detection, and appropriate engineering and packaging so as to be reliable (our overall hardware reliability has been excellent). 32MB of ECC memory is packaged in a 12.5 inch rack mounted unit, including power supplies, control, and cooling. A control bus, the "MEMBUS", provides space for up to 3 control and interface cards. No provision is made for daisy-chaining the memory arrays, so our interface addresses the issues of building an expandable system.

The FPS-164 I/O bus is a 64-bit, 181-ns synchronous bus which allows memory accesses to be started by a device as frequently as every cycle, for a bandwidth of 44 MB/s, subject to restrictions on memory bank conflicts and refreshes. A programmed I/O and interrupt capability provides the CPU control over and synchronization with I/O devices.

FPS provides host adapters for the DEC Unibus (VAX hosts), Multibus (Apollo hosts), and the IBM channel bus. The only mass store option offered currently is the D64 disk subsystem, which uses a FPS-164 I/O bus to Unibus adapter and a Unibus disk controller. This scheme has peak bandwidth limited to approximately 2MB/s, regardless of the disk being used.

It was clear when we started that we would have to build an FPS I/O bus adapter to even approach the required data rate. Floating Point Systems made this project feasible by providing us an excellent writeup on the I/O bus[2].

3.3. What we Implemented

We built a pair of circuit boards with interconnect cables which together comprise the interface. The master card, installed in the FPS-164 I/O bus, drives a cable bus connecting up to 6 of the slave cards, each installed in a System/3000 holding up to 32MB of memory. Together the master and slave use approximately 320 S-TTL chips.

The master card and all transactions occurring on the cable bus are clocked by the FPS-164 master clock. Each slave card decouples MEMBUS timing from 164 timing with a 16x66 bit FIFO. Parity is carried through the system from the MEMBUS through the cable and bus drivers in the

master card. Extensive support for diagnostics and error analysis is provided and, thankfully, has been little used.

We have built and placed in regular operation one master with one slave 32MB System/3000.

3.4. Software Interface

Programs running on the FPS-164 initiate transfers through a set of control registers accessed with FPS-164 I/O instructions. When a transfer is initiated, the program places the FPS-164 memory address, Sys/3000 box number, Sys/3000 memory address, word count, and transfer direction in the device control registers. In the case of reads from memory, after a latency period of 1 μs during which the FIFO fills, a word is transferred to FPS-164 main memory every cycle; done is signaled by a bit in the status register which can generate an interrupt.

Since this I/O device consumes all the memory bandwidth of the FPS-164 while it is transferring, it has not been our practice to use interrupt-driven software to attempt to overlap computation and I/O; the program waits for I/O to complete by initiating a memory reference, which will hang the CPU until the mass memory finishes transferring, and then checking the device status register.

3.5. Further Features and Explorations

Simultaneous transfers by the FPS-164 and a secondary device (using another MEMBUS control slot) are possible. The program can issue 1-word reads and writes addressed to control registers in an interface controller plugged into another slot in the MEMbus in any System/3000. That interface has the ability to act as a "secondary MEMbus master" and access Sys/3000 memory independently of the FPS-164, and to interrupt the 164 via an interrupt status line passed through the cable bus.

This capability allows us to plug a disk subsystem directly into the Sys/3000, control it from the 164, and experiment with using the Sys/3000 as a "disk cache" or "staging memory" in the handling of very large out-of-core problems.

This same mechanism could be used to share high-speed memory with other general-purpose or specialized processors. Transfer bandwidth is statically allocated in the current design so as to allow only approximately 6MB/second transfer rates by the secondary controller, but this could easily be rectified.

The Sys/3000 controller generates a rigid increasing or decreasing MEMbus address sequence during block transfers, and our interface does the same on the FPS-164 side. Certain algorithms would benefit from other, easily generated addressing patterns; matrix transposition is an example.

We are considering adding an "address skip count" to provide more addressing flexibility for these problems.

4. Bulk Memory Software

An important consideration in any interface is that it be as transparent to the user as possible, and that programs making use of it be as transportable as possible. Because the bulk memory is significantly faster at block transfers than at scalar transfers, it is best to look at it as a fast secondary memory. Secondary memory is supported through FORTRAN by unformatted reads and writes to (usually scratch) files. In fact, many existing large applications make use of this facility to achieve high performance out-of-core codes. Our bulk memory interface is designed to be completely transparent at the FORTRAN level. Both permanent and scratch files are supported. The difference to the programmer of files on the bulk memory from files on the disk is the file's *device number*, which is part of the filename specification in the FORTRAN for the FPS-164. Thus, with a single change in the file name, programs which use unformatted I/O to disk can be converted to using unformatted I/O to bulk memory.

Formatted reads and writes are not supported as the inefficiency of formatted I/O eliminates most of the advantages of the bulk memory. FORTRAN's internal file facility provides a way for the user to format their data before writing it or after reading it, should formatted I/O be needed.

4.1. Bulk Memory File System

The Bulk Memory software interface supports all unformatted I/O defined in the FORTRAN 77 standard [1], including the record specifier `REC` in read and write statements, the opening and closing of `NEW`, `OLD`, `UNKNOWN`, and `SCRATCH` files, and `IOSTAT` queries. All system dependent information is, following the practice of the compiler for the FPS-164, encoded in the file name field. This information includes the device, file size, and a file name of up to 16 characters. Devices include disk and bulk memory; the bulk memory is assigned to a non-existent disk controller. File extents are not allowed in the bulk memory. Reads and writes are executed as single block transfers for each item in the I/O list; as Figure 1 shows, for any but the smallest items, the FORTRAN I/O version is as fast as using low level, non-portable, subroutine calls.

4.2. Roll-in/roll-out

The only time-sharing mechanism in the current release of the FPS-164 software is swapping. At the end of a quantum, if there are users waiting to use the FPS-164, the current process's state is saved to disk and a new process is swapped in. There is no paging on this machine, and processes

are not allowed to coreside in memory. For large programs, the entire memory, in our case 4 MB, may have to be written to disk, at a rate of approximately .3 MB/second. We measured the time spent rolling in and out a 4MB job to the D64 subsystem at approximately 13 seconds. Therefore, reasonable quantums required by the standard roll-in/roll-out are on the order of 5 to 15 *minutes*, which makes time-sharing impractical. We have modified the roll-in/roll-out procedure to write to a reserved section of the external memory instead of the disk in order to substantially reduce the swapping time and make pseudo time-sharing realizable.

4.3. Bulk Memory Management Tools

Since we view the Bulk Memory as a disk, we need some basic file system tools. We have built

DELETE	Deletes files.
DIR	Gives a directory of files.
TODISK	Move a file from the bulk memory to the disk.
TOBULK	Move a file from the disk to the bulk memory.
BULKDIAG	Bulk memory diagnostics

5. Benchmarks

The I/O bandwidth of the external memory subsystem is 41 MB/second for sufficiently large transfers. As we described in Section 3, the interface is designed for efficiency in large data transfers. We first examine how the transfer rate varies with the size of the block of data transfered. In Figure 1, we graph the μs/byte for transfers of blocks of data of varying size. We find that as the size of the transfer increases, the time/byte asymptotically approaches approximately .0234 μs. We compare this to times published by FPS for the FPS-164 D64 disk subsystem[3]. Our experimental time using unformatted FORTRAN I/O on the D64 subsystem is 3.14 μs/byte for transfers of large arrays, which is substantially longer than the 1.12 μs/byte derived from the FPS times using low level I/O. We also include estimated times for a very fast disk, a new drive from Fujitsu, which has a seek/latency time that varies between 20-30 ms and a transfer rate of .1111... μs/byte. We graph the data using an average seek time of 25 ms. However, even with totally asynchronous I/O programmed to eliminate seek time, the Fujitsu disk has a transfer rate that is 4.7 times longer than that of the external memory subsystem.

As a means of comparing the external memory with the FPS-164's D64 disk subsystem, we ran a simple power method benchmark. We applied ten iterations of the power method to solve

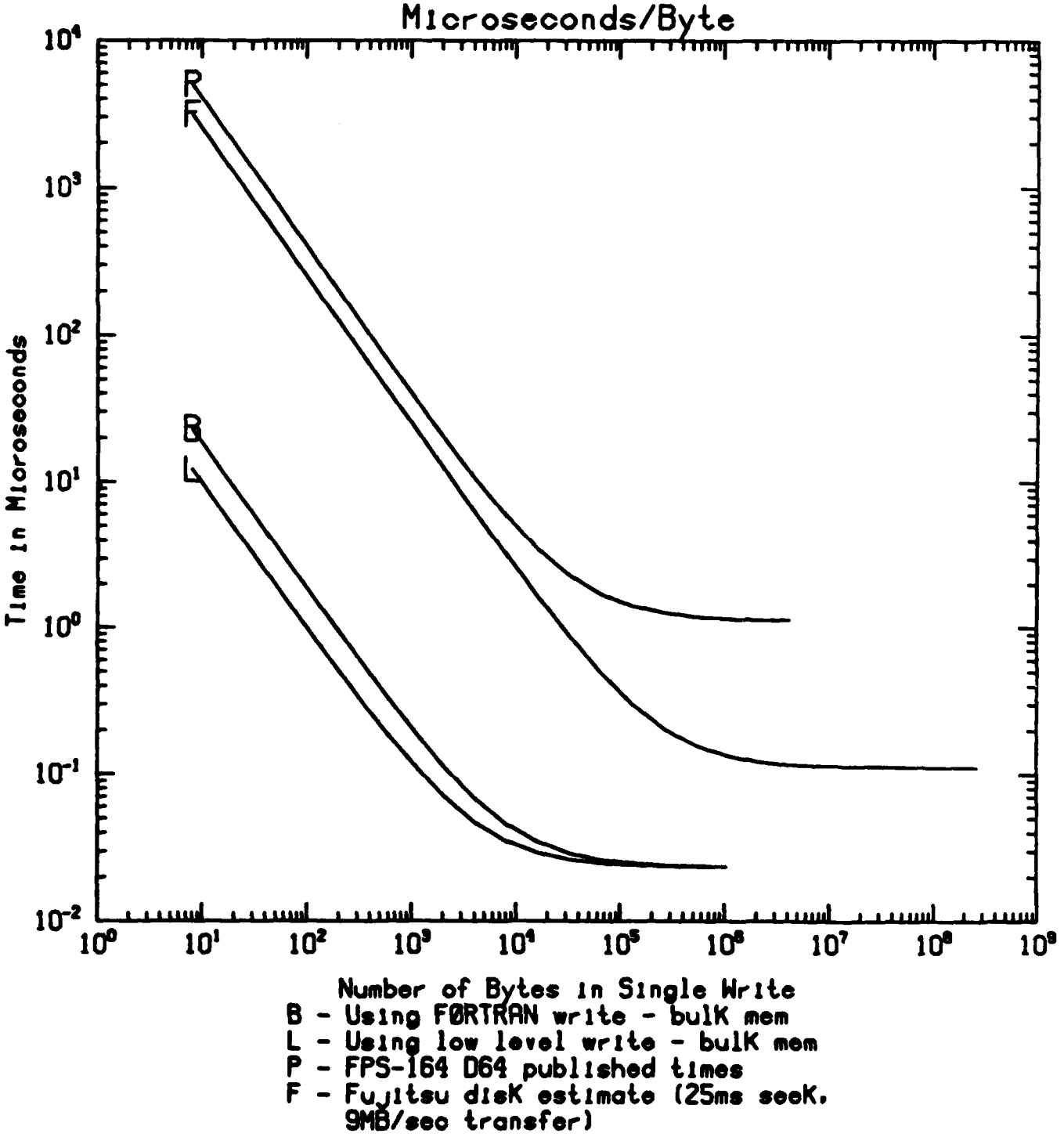


Figure 1: Comparison of transfer rates for bulk memory and disk.

for the dominant eigenvalue and eigenvector of a matrix of size 1024 by 1024. This performs 10 dense vector matrix products, where the vector resides in memory and the matrix on the external storage device. We used a block transfer size of 32768 words. Both wall clock and cycle time measurements were taken, as the cycle counter does not properly track time during disk I/O. We subsequently ran the same problem on the Cray-1, using both disk and SSD. The computation was not optimized on the Cray, so that emphasis is on the I/O bandwidth rather than a Cray/FPS-164 computation comparison. We present the resulting times in Table 1. The I/O bandwidth data for the disks include seek and latency time.

	FPS-164	Cray
Computation excluding I/O	4.9	1.3
Solution with external mem	7.3	2.7
Bandwidth (MB/sec) of external mem	41.2	68.0
Solution with disk	315.0	28.9
Bandwidth (MB/sec) of disk	.3	3.3

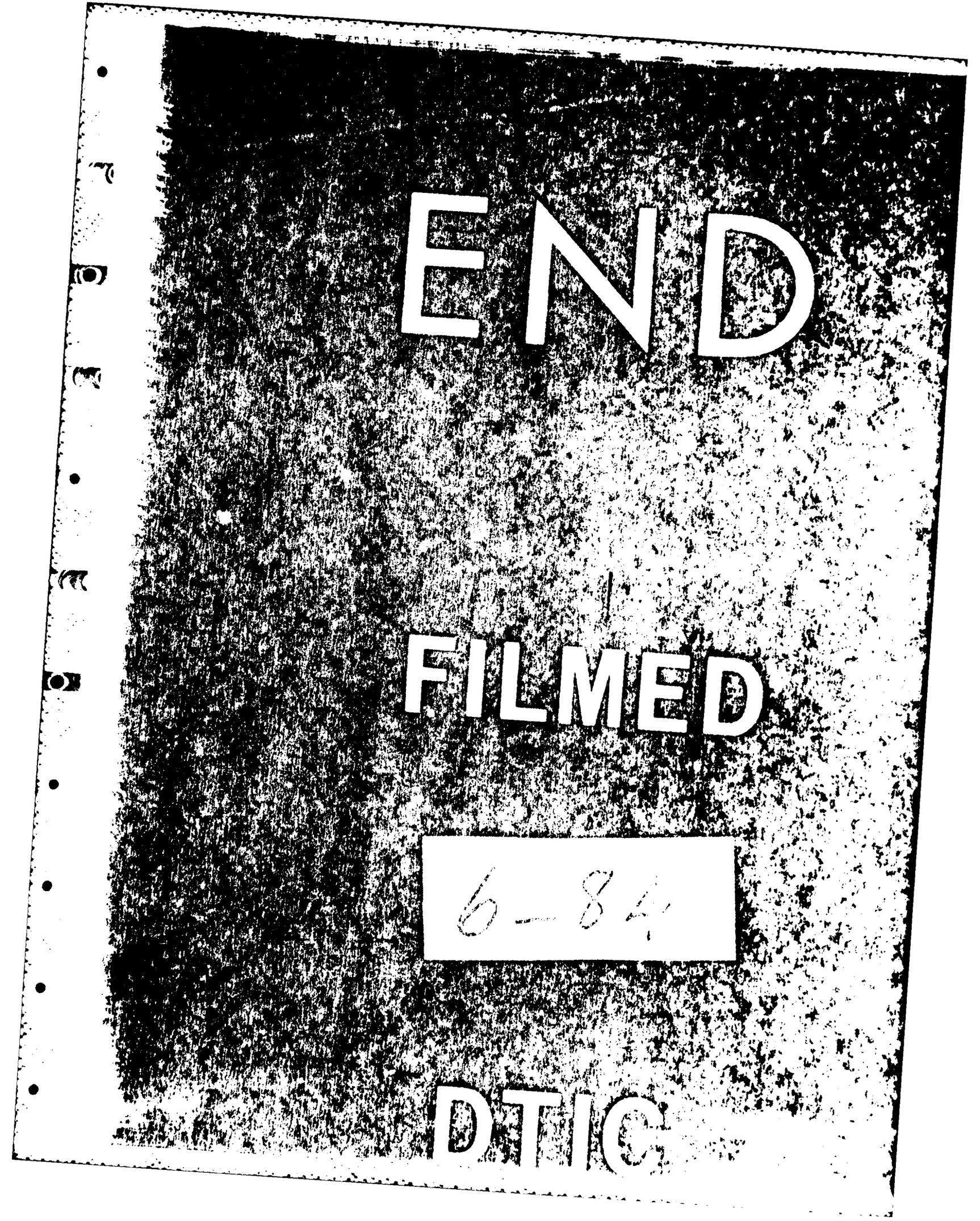
Table 1: Time in seconds for solution of power method benchmark.

References

[1] *American National Standard Programming Language FORTRAN*, American National Standards Institute, Inc., 1978.

[2] FPS-164 Host Interface Hardware Designer's Guide, 1982.

[3] FPS-D64 Disk Subsystem Reference Manual, 1981.


END
FILMED
6-84
DTIC